

AFRL-IF-RS-TR-2005-318
Final Technical Report
September 2005



BETTER FAULT TOLERANCE VIA APPLICATION ENHANCED NETWORKS

University of Arizona

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J903

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-318 has been reviewed and is approved for publication

APPROVED: /s/

DAVID E. KRZYSIAK
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE			<i>Form Approved</i> <i>OMB No. 074-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE SEPTEMBER 2005	3. REPORT TYPE AND DATES COVERED Final Jun 00 – Jun 03	
4. TITLE AND SUBTITLE BETTER FAULT TOLERANCE VIA APPLICATION ENHANCED NETWORKS			5. FUNDING NUMBERS C - F30602-00-2-0560 PE - 62702F PR - J903 TA - 01 WU - A1	
6. AUTHOR(S) John Hartman, Scott Baker, William Evans, Gregg Townsend, David Kirkpatrick				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Arizona Department of Computer Science Tucson Arizona 85721			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency AFRL/IFGA 3701 North Fairfax Drive 525 Brooks Road Arlington Virginia 22203-1714 Rome New York 13441-4505			10. SPONSORING / MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2005-318	
11. SUPPLEMENTARY NOTES AFRL Project Engineer: David E. Krzysiak/IFGA/(315) 330-7454/ David.Krzysiak@rl.af.mil				
12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.				12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 Words) This effort developed, implemented, and evaluated a set of mechanisms that protects individual network nodes from faults and denial of service. Uses active network technology to build networks, systems, and applications that tolerate faults and denial of service attacks.				
14. SUBJECT TERMS Communications Networks, Fault Tolerance, Computer Networks, Denial of Service, Survivability				15. NUMBER OF PAGES 41
				16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Table of Contents

1.0 Project Description.....	1
1.1 Local Resource Management.....	1
1.2 Distributed Terrain Navigation.....	1
1.3 Network-Resident Storage.....	1
2.0 Right-Triangulated Irregular Networks	2
2.1 Introduction.....	2
2.2 Related Work	5
2.3 Sub-grid and TIN	5
2.4 Sub-grid and TIN hierarchies.....	6
3.0 RTIN Hierarchy	7
3.1 Data structure	10
3.2 Reducing Space Usage.....	13
3.3 Neighbor Calculation	14
3.4 On-the-fly Surface Approximation	16
3.5 Experimental Results	19
3.5.1 Single surface performance.....	19
3.5.2 Hierarchical Performance	24
3.6 Comparison to Previous Work.....	26
3.7 Conclusions.....	26
4.0 Mirage NFS Router.....	27
4.1 Mirage Introduction	27
4.2 NFS	29
4.3 Mirage	31
4.4 Virtual Server Abstraction	31
4.5 Virtual Handle Mapping	33
4.6 Mirage Crash Recovery	34
4.7 Conclusion	34

Table of Figures

FIGURE 1: SPLITTING TRIANGLE T	8
FIGURE 2: PIECES OF LAVES NETS OF TYPE $[4^4]$, $[6^3]$, $[4.6.12]$, AND $[4.8^2]$	9
FIGURE 3: NON-UNIFORM PARTITION BASED ON THE $[4.6.12]$ LAVES NET	9
FIGURE 4: THE POSSIBLE TRIANGLE ORIENTATIONS AND THEIR LEFT AND RIGHT CHILDREN. LEFT AND RIGHT ARE DENOTED BY 0 AND 1 RESPECTIVELY.....	11
FIGURE 5: THE BINARY TREE REPRESENTATION OF A RTIN PARTITION.	11
FIGURE 6: THE i -NEIGHBORS OF A TRIANGLE. IN THIS CASE, ALL NEIGHBORS ARE THE SAME SIZE AS THE TRIANGLE. THE NUMBERS WITHIN THE INNER TRIANGLE ARE THE VERTEX NUMBERS OF THE INNER TRIANGLE.	14
FIGURE 7: USE OF THE RECURSIVE NEIGHBOR EQUATIONS TO CALCULATE THE 2-NEIGHBOR OF 0111.....	15
FIGURE 8: C-CODE FOR FAST TRIANGLE NEIGHBOR CALCULATION	17
FIGURE 9: MAXIMUM ERROR VERSUS NUMBER OF POINTS IN APPROXIMATION. CRATER LAKE DATA SET.	19
FIGURE 10: ROOT MEAN SQUARED ERROR VERSUS NUMBER OF POINTS. CRATER LAKE DATA SET.	21
FIGURE 11: MAXIMUM ERROR VERSUS MEMORY USAGE.....	21
FIGURE 12: ROOT MEAN SQUARED ERROR VERSUS MEMORY USAGE CRATER LAKE DATA SET.	23
FIGURE 13: THE AVERAGE TIME TO CONSTRUCT AND TIME TO CONSTRUCT AND DRAW A SURFACE APPROXIMATION OF MOUNT ST. HELENS AS A FUNCTION OF THE ERROR PER DISTANCE TOLERANCE. THE GRAPH ALSO SHOWS THE NUMBER OF TRIANGLES (IN 1,000s) IN THE APPROXIMATION.	23
FIGURE 14: THE AVERAGE TIME TO CONSTRUCT AND TIME TO CONSTRUCT AND DRAW A SURFACE APPROXIMATION OF MOUNT RAINIER EAST AS A FUNCTION OF THE ERROR PER DISTANCE TOLERANCE. THE GRAPH ALSO SHOWS THE NUMBER OF TRIANGLES.	24
FIGURE 15: FULL DETAIL, PRE-1980 MOUNT ST. HELENS (305K TRIANGLES)	25
FIGURE 16: FULL DETAIL, POST-1980 MOUNT ST. HELENS (305K TRIANGLES).....	25
FIGURE 17: TOLERANCE=1M PER KM (44,500 TRIANGLES).....	25

FIGURE 18: TOLERANCE=1M PER KM (39,800 TRIANGLES).....	25
FIGURE 19: TOLERANCE=4M PER KM (10,600 TRIANGLES).....	25
FIGURE 20: TOLERANCE=4M PER KM (10,200 TRIANGLES).....	25
FIGURE 21: MIRAGE ROUTES REQUESTS AND REPLIES BETWEEN THE NFS CLIENTS AND THE NFS SERVERS.	28
FIGURE 22: SEQUENCE OF NFS CLIENT REQUESTS AND SERVER REPLIES TO READ THE FIRST 2048 BYTES OF /HOME/FRED/PHOTO.	30
FIGURE 23: THE SET OF MOUNT POINTS EXPORTED BY THE MIRAGE VIRTUAL SERVER IS THE UNION OF THE SETS OF MOUNT POINTS EXPORTED BY THE INDIVIDUAL NFS SERVERS.....	31
FIGURE 24: THE VIRTUAL SERVER EXPORTS FILE SYSTEMS WHOSE NAMESPACES MERGE THE NAMESPACES OF THE UNDERLYING EXPORTED FILE SYSTEMS.....	32
FIGURE 25: THE MIRAGE ROUTER MAPS BETWEEN THE VIRTUAL FILE HANDLES USED BY THE CLIENTS AND THE PHYSICAL FILE HANDLES USED BY THE NFS SERVERS.....	33
FIGURE 26: FORMAT OF A VIRTUAL FILE HANDLE. OFFSETS ARE IN BYTES.	33

1.0 Project Description

Our goal was to advance the use of active network technology to build networks, systems, and applications that tolerate faults and denial-of-service attacks. Active networks allow high-level functionality to be embedded in the network, improving the network's ability to detect and react to failures and attacks. Rather than provide generic fault-tolerant protocols or middleware, the network can handle attacks and faults in an application-specific fashion. We call this embedding of application-specific fault handling *application-enhanced fault tolerance*. We pursued three main research thrusts:

1.1 *Local Resource Management.*

Each node of an active network contains resources that legitimate applications can exploit, but that malicious applications can attack. Faults and denial-of-service attacks can be viewed as a resource allocation and scheduling problem. As part of our research we developed techniques for accurately accounting for all of a node's resources, and scheduling algorithms that allocate multiple resources to the code in a meaningful way.

1.2 *Distributed Terrain Navigation.*

This application supports the collection of terrain data from servers, and its dissemination to clients that are navigating or visualizing the terrain. The system embeds terrain-navigation-specific functionality into the network, enabling the system as a whole to tolerate denial-of-service attacks and faults better.

1.3 *Network-Resident Storage.*

We investigated the benefits of persistent storage within the network. The resulting *network-resident storage* system has several benefits over existing architectures, in terms of performance, fault tolerance, and resistance to attacks. Many distributed storage system functions, such as data replication, reconstruction, and synchronization are also best handled in the network because they involve filtering and merging network traffic, thus also improving the system's overall performance.

2.0 Right-Triangulated Irregular Networks

2.1 *Introduction*

We describe a hierarchical data structure for representing a digital terrain (height field) which contains approximations of the terrain at different levels of detail. The approximations are based on triangulations of the underlying two-dimensional space using right-angled triangles. The methods we discuss permit a single approximation to have a varying level of approximation accuracy across the surface. Thus, for example, the area close to an observer may be represented with greater detail than areas which lie outside their field of view. We discuss the application of this hierarchical data structure to the problem of interactive terrain visualization. We point out some of the advantages of this method in terms of memory usage and speed.

In this report we describe a method for approximating a surface which is presented to us as a two-dimensional array of height values. We assume that the height value in location i, j of the array is the true height of the surface at location x_i, y_j where x_i and y_j are easily derived from the indices i and j . Thus we ignore questions of error in the input. For our purpose, the surface is defined by the values in the height array. Of course, this leaves most of the surface, those points with x, y coordinate not equal to x_i, y_j for some i, j undefined. We make no assumptions about these intermediate points. We measure the goodness of our approximations only against the elevations in the array.

Such surface representations are common and can be quite large. For example, a 10 meter resolution 7.5 minute digital elevation model (DEM) of Mount Rainier, Washington has $977 \times 1405 = 1,372,685$ data points. Combining several of these map sheets, which is necessary, for example, in accurate watershed calculation, can result in arbitrarily large data sets.

Computations involving such data sets are very expensive. Both memory and time used by the analysis algorithm can be prohibitively large. In such a case, we cannot use the entire surface representation. Either the surface must be subdivided into manageable pieces, which only works if the analysis can be localized to these pieces, or we must use an approximation to the surface in place of the original. Of course, the approximation should be constructed so that the answers we obtain in analyzing it approximate the answers we would obtain for the true surface.

Surface approximation of regular gridded DEMs typically possesses one of two forms. Either the approximation is itself a regular gridded DEM, a regular *sub-grid* of the original data, or it is a triangulated irregular network (TIN), a possibly irregular subset of the original data points whose projection into the x, y – plane is triangulated.

The approximation form we describe in this paper lies somewhere between the regular sub-grid and the irregular TIN. It is more regular than the TIN and more flexible than the sub-grid. Like the TIN, it is a triangulated subset of the original data points, but, unlike the TIN, the subset is not arbitrary. The subset is such that all triangles are right angled and isosceles. Consequently, we call this form of approximation a right-triangulated irregular network (RTIN).

The main benefit of the RTIN approximation is not in providing a single approximation to a surface, but rather in providing a framework of many approximations at varying levels of detail.

A given approximation may be well suited to one type of approximation but not another. An approximation that represents ridge lines very accurately may not be the best approximation for determining wildlife habitat. In order to form an appropriate approximation to the surface, we need to know for what purpose the approximation is intended. It is time consuming and often impractical to create and store an approximation for each intended application. Often a better solution is to devise a general framework for approximation that can yield approximations tailored to particular applications. The problem then becomes calculating the appropriate approximation from the framework. Extracting an approximation from an existing framework has the potential advantage of providing tailor-made approximations much more rapidly than constructing a suitable approximation from scratch. Of course, this works well only if the framework can provide an appropriate approximation quickly. This is critically important when many different approximations are required in rapid succession.

The framework we propose in this report is a hierarchy of approximations. It is a hierarchy in that the framework contains a range of representations, from coarse to fine. However, the representations are not segregated. The hierarchy can provide a single approximation that mixed coarse and fine level representations. This allows us to obtain high levels of detail in certain regions of the surface without forcing us to represent the entire surface at this high resolution. For example, in order to accurately approximate elevation, mountainous regions may require much finer detail than flat plains.

An additional benefit to having the entire hierarchy of approximations is that it can make tasks such as point location faster than they would be if one had only a single approximation.

The disadvantage to keeping a hierarchy of approximations is that it requires more storage than a single approximation. One of the main goals of this work is to demonstrate a practical and space efficient approximation hierarchy. Our implementation of the RTIN hierarchy requires less than 14 bytes per original data point. This includes two bytes for the height value itself. Though seven times more expensive than the original data set, the hierarchy provides an extensive range of approximations.

To illustrate the utility of the RTIN hierarchy, we describe its implementation for interactive visualization of a surface. The goal of interactive visualization is to show what a user would see as they “fly” or “walk” over the surface. The problem is that the surface is too detailed to render at full resolution at a reasonable speed. Thus, the surface must be approximated in order to reduce the time needed to update the display. A single approximation, independent of the eye position, could be used, but the resulting image may be unacceptably coarse in areas close to the eye. Our approach is to adapt the approximation to the location of the eye so that regions close to the viewer are a high resolution while distant or non-visible regions are more coarsely represented.

The demand for varying levels of detail within one approximation must be answered carefully. Areas of high detail must match with adjacent areas of low detail in such a way that the surface remains “continuous”. Discontinuities are very noticeable in interactive applications.

Interactive visualization is a good test of our hierarchy since it requires many of the desired properties: we must be able to choose an appropriate approximation rapidly; the approximation must allow different levels of detail at different locations on the surface; and the surface must be a continuous surface after interpolation (gaps in the surface can be glaringly obvious).

The focus of this paper will be on describing the RTIN hierarchy and the data structure used to implement it. Later sections of the paper will detail the performance of the hierarchy both as a solution to the interactive visualization problem and as a way to represent a single (non-hierarchical) surface approximation.

We start by reviewing a portion of the large body of work that has been done on surface approximation. We then review the two popular surface representations, the sub-grid and the TIN, and describe hierarchies based on these representations. This sets the stage for our discussion of the RTIN hierarchy of approximations.

2.2 Related Work

A great deal of work has been done on the approximation of surfaces, and, in fact, on the particular problem of multi-resolution surface modeling of which hierarchies are one type. A recent survey by De Floriani, Marzano, and Puppo discusses many of the advances in this field. This survey does not refer to a large body of more recent work that has appeared in SIGGRAPH96 and EuroGraphics96, both of which had sessions on multi-resolution surface modeling. A recent tutorial by Puppo and Scopigno covers this more recent work as well as a wide range of surface simplification techniques.

The most relevant work to our approach that has appeared in the surface approximation literature is the work by Lindstrom et al and Puppo. Both independently discovered hierarchies similar to the one discussed in this paper. We discuss and compare their approaches to ours.

Another source of related work comes from the study of general lattice structures. Bell et al. cite work on crystal lattice structures by Subnikov in 1916 and Laves in 1931 that defined various planar partitions including the one upon which our hierarchy is based. These partitions have been studied as a means of addressing spatial regions with application to image encoding and adaptive mesh generation for finite element solvers in two and higher dimensions. Hebert appears to be the first to describe non-uniform versions of the particular partition discussed in this paper, though the concept is closely related to the quad-tree data structure discussed by Samet and the restricted quadtree of Von Herzen and Barr.

2.3 Sub-grid and TIN

Approximations of height data typically come in one of two types: sub-grid or TIN. A sub-grid is of the same form as the input array of height values. It is an array of height values at regularly spaced x and y coordinates of a height value from its array indices. Thus, the only storage needed is for the height or z values of the sample points. To obtain the height at x, y coordinates not in the sample set, one uses some form of interpolation. In fact, what is often done is that the four sample points forming the smallest square containing the x, y position are used to determine the height at x, y via bilinear interpolation. The regular spacing of the sample points makes it easy to determine the closest sample points to a given x, y position. Alternatively, one might choose to use linear interpolation based on a triangulation of this smallest square (arbitrarily picking one of the square's two diagonals to split it into two triangles).

The sample points of a Triangulated Irregular Network (TIN) are an arbitrary subset of the original data points. In order to specify this subset, the x, y coordinates (i.e. the indices in the input array) of each sample point in the TIN must be stored explicitly, in addition to the z value. The sample points, without their height component, are triangulated (typically using a Delaunay triangulation) and linear interpolation is used to obtain the height at an arbitrary point (x, y) from the heights of the vertices of the triangle which contains the point.

The advantage of the TIN is that the sampling can be non-uniform: large featureless regions can be sampled at a coarser resolution than regions with a great deal of variation. The disadvantage is that a TIN requires more storage for the same number of sample points: a 3:1 ratio if x, y and z values require the same precision. The storage disadvantage becomes worse if one requires adjacency information for the triangulation; if for example, one needs the ability to traverse the surface approximation from triangle to adjacent triangle.

The factor of three differences in storage requirements has led to an unfavorable view of TINs. Kumlér argues that to achieve a certain degree of accuracy in the representation of a height field, sub-grids require less storage space than TINs. However, he then states, "I did not expect these results. My intuition led me to believe, and I continue to believe, that the irregular TIN model is, in general, a more efficient method for representing an irregular natural surface." He then goes on to explain his conviction by claiming that methods to construct more efficient TINs will be developed in the future. Our results comparing error measures to space usage for sub-grids and TINs may shed some light on this issue.

2.4 Sub-grid and TIN hierarchies

Both sub-grid and TIN approximations may be organized into hierarchies. A hierarchy of sub-grid approximations is obtained by varying the spacing of the regularly sampled grid points. For example, one level of the hierarchy may contain every other sample point from the input array while a coarser level may contain every fourth sample point. Typically, each level of the hierarchy is a regularly sub-sampled approximation of the next finer level. Thus, the amount of storage needed for the entire hierarchy is just the amount needed for the finest level of approximation. The price of such small storage is the rigidity and uniformity of the sampling pattern.

A TIN hierarchy allows varying levels of detail within an approximation at the expense of more storage. There are several hierarchies based on irregular triangulations. One may roughly divide them into those that preserve the boundaries of coarse triangulations in going to finer triangulations, and those that do not. In the first case, the hierarchy is organized in a tree structure; each node

represents a region, and the children of a node represent the regions into which the node's region is partitioned in going from one level to the next finer. Refinement is typically performed by choosing within each region a point from the original input data, adding it to the set of sampled points, and re-triangulating (maintaining the previous boundaries). Such a scheme tends to create long, thin triangles which may provide a less accurate approximation than shorter-edged, larger-angled triangles. On the other hand, any level of detail can be achieved in one area of the triangulation without affecting other areas. In terms of the tree structure, any sub-tree containing the root represents a valid surface approximation.

In the second case, one can choose to re-triangulate at each level using, for example, Delaunay triangulation. In this case, it is not in general possible to refine a particular region independent of other regions, i.e. combining pieces from different levels of the hierarchy is difficult. DeBerg and Dobrindt recently showed how a hierarchy of Delaunay triangulations which shows local refinement can be constructed. This is accomplished using a more complicated structure than the simple tree used in the first case. DeBerg and Dobrindt cite memory usage of 132 bytes per data point.

Our goal is to provide a hierarchy of surface approximations which is a compromise between the sub-grid and the TIN approximations. We hope that a more regular "grid-like" TIN will combine the sub-grid's advantage of smaller space per vertex with the TIN's advantage of local adaptability to terrain.

3.0 RTIN Hierarchy

Our hierarchical surface representation contains partitions, called RTIN partitions, of the two dimensional square into right-angled, isosceles triangles. Such partitions have been studied in relation to crystal lattices, image encoding, and in fact, geographic visualization systems. The partitions contained in the hierarchy are defined inductively as follows. The partition composed of two triangles formed by dividing the square from northwest to southeast corner is the coarsest partition in the hierarchy (i.e. the one containing the largest triangles). From a partition in the hierarchy, a new, more detailed partition may be formed by splitting any one of the non-terminal triangles in the partition, where a non-terminal triangle is one which is larger than some fixed threshold depending on the resolution of the underlying height field. A triangle T , is split by adding an edge from its right-angled vertex to the midpoint of its hypotenuse. The resulting partition may be non-triangular meaning that the new vertex introduced at the midpoint of the hypotenuse may lie on the border of a triangle R (T 's neighbor across the hypotenuse) turning R into a quadrilateral. This represents a problem when using the partition to create a continuous surface via linear interpolation. Each vertex in the partition has a height value defined by the height field. Since R is a quadrilateral, linear interpolation over R is not well-

defined, and performing linear interpolation using R 's original three vertices may cause gaps in the surface. To avoid this problem, we continue or propagate the split into R . If R is larger than T then the hypotenuse of T forms a leg of R and we first split R (perhaps propagating this split) and then split the resulting triangle which shares its hypotenuse with T . If R is the same size as T we need only split R (no further propagation is necessary). See figure 1.

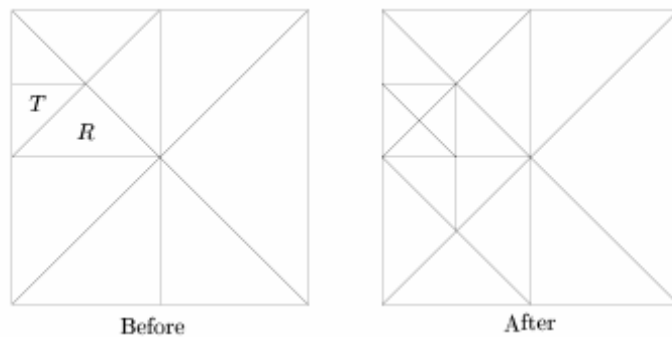


Figure 1: Splitting triangle T

Propagation means that a single split operation can cause more than one triangle to split. However, splitting a triangle T cannot cause a triangle smaller than T to be split. Thus, only a finite number of triangles will be split in a single split operation, and the operation is guaranteed to terminate. In fact, at most two triangles of each size equal to or larger than T and no triangles smaller than T will split as a result of splitting T .

The most detailed partition in the hierarchy is one in which no triangle may be split, i.e. all triangles are terminal. The most detailed partition is uniform: every triangle is the same size. The height of the (i, j) th grid point is the height of the (i, j) th sample point in the height array. The integer k (or, equivalently, the threshold defining the terminal triangle) is chosen so that the grid is large enough to contain the height array.

If extended to cover the plane, such a uniform partition is called a $[4.8^2]$ Laves net named by Bell, Diaz, Holroyd, and Jackson after F. Laves. The name lists the degree of the vertices of the atomic polygon within the partition, in cyclic order around the polygon. In this case, the right angled vertex has degree 4 and the two other vertices both have degree 8.

Other Laves nets which may be used as the basis for single shape hierarchies are the square $[4^4]$, the equilateral triangle $[6^3]$, and the 30-60 right triangle $[4.6.12]$. See figure 2. These nets all share the crucial property that they are infinitely refinable using similar polygons: a given polygon in any net can itself be partitioned using polygons of the same shape.

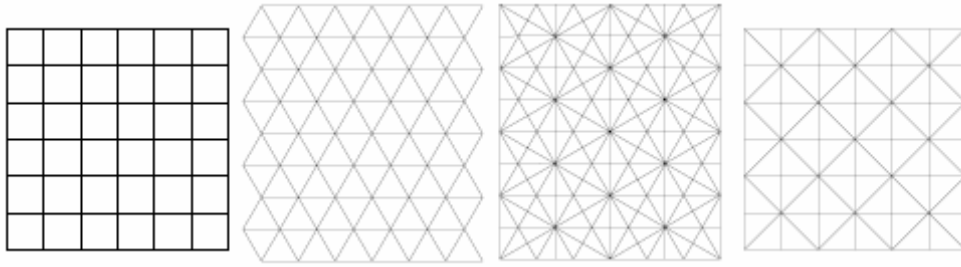


Figure 2: Pieces of Laves nets of type $[4^4]$, $[6^3]$, $[4.6.12]$, and $[4.8^2]$

Unlike a Laves net, a partition may be non-uniform. Only certain Laves nets can form the basis of non-uniform partitions. For partitions based on the square or equilateral triangle net, refining one region forces all other regions in the partition to be refined. This refining is forced in order to maintain the basic shape of the regions in the partition. For example, refining one square in a square net introduces new vertices on the edges of adjacent squares which forces these squares (now pentagons) to be refined which forces further refinement until the partition is uniform. The equilateral triangle net behaves in a similar manner. Thus these nets, without special compensation for non-similar regions, cannot be the basis of a non-uniform partition. As mentioned earlier, refining (of splitting) a triangle affects at most two triangles of each size. See figure 3.

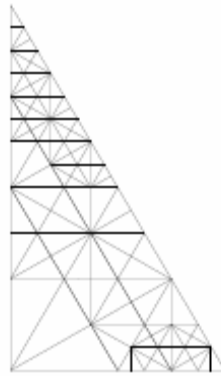


Figure 3: Non-uniform partition based on the $[4.6.12]$ Laves net

Unlike a Laves net, a partition may be non-uniform. Only certain Laves nets can form the basis of non-uniform partitions. For partitions based on the square or equilateral triangle net, refining one region forces all other regions in the partition to be refined. This refining is forced in order to maintain the basic shape of the regions in the partition. For example, refining one square in a square net introduces new vertices on the edges of adjacent squares which force these squares (now pentagons) to be refined, which forces further refinement until the partition is uniform. The equilateral triangle net behaves in a similar manner. Thus these nets, without special compensation for non-similar regions, cannot be the basis of a non-uniform partition. The $[4.6.12]$ (30-60 right triangle) partition

and the $[4.8^2]$ (isosceles right triangle) partition do not suffer this limitation. As mentioned earlier, refining (or splitting) a triangle affects at most two triangles of each size in the $[4.8^2]$ partition. In the $[4.6.12]$ partition, the number of triangles affected of each size equal to or larger than the triangle initially split is at most 12. See figure 3.

Three properties favor the $[4.8^2]$ partition for hierarchical representation of surface approximations. First, it is a triangular subdivision which means that height values in the interior of the triangle can be linearly interpolated from the height values at the vertices of the triangle in an unambiguous way. The benefits of linear interpolation over other methods of interpolation (such as bilinear or inverse-distance weighting) are speed and simplicity. For 3D visualization, linear interpolation is supported in hardware on some machines, making it the only reasonable interpolation scheme for large numbers of triangles.

Second, the vertices of a uniform $[4.8^2]$ partition are equally spaced grid points, in contrast to a $[4.6.12]$ partition. Thus, each vertex represents a sample point from the original data set, and every point from the data set is included in the most detailed partition.

Third, one area of the partition can be refined, while maintaining the “no gap” or surface property, without affecting a large number of regions. Refining or splitting a triangle affects only a small number of other triangles in contrast to the $[4^4]$ or $[6^3]$ partitions.

3.1 Data structure

The previous section introduced RTIN partitions. We now discuss how these partitions may be represented in a data structure both individually and collectively as a hierarchy. The general idea is suggested by the definition of the RTIN partition: since each triangle may be divided into two similar pieces, a partition is represented by a binary tree. In particular, each (triangular) region in the partition is represented by a leaf in the binary tree that represents the partition.

The root of the tree represents the initial square (the only case in which the region associated with a node is not a right triangle). The children of the root correspond to the regions obtained by dividing the square along its northwest-southeast diagonal. The left (right) child corresponds to the triangle containing the southwest (northeast) corner. In general, the children of a node with triangular region t correspond to the triangles obtained by “splitting” T from the midpoint of its hypotenuse to its right-angled vertex. The left (right) child is the triangle to the left (right) of this split (looking from the midpoint to the vertex). See figure 4.

This establishes a labeling scheme for the triangular regions in a RTIN partition. The label of a region is a description of the path from the root to its corresponding node in the binary tree. The path description is the concatenation of the labels of the edges on the path from the root to the node, where an edge is labeled 0 (1) if it leads to a left (right) child. An example of a RTIN partition and its representative binary tree are shown in figure 5.

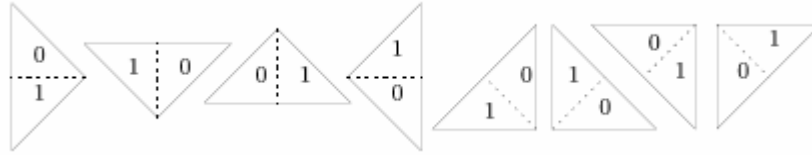


Figure 4: The possible triangle orientations and their left and right children. Left and right are denoted by 0 and 1 respectively.

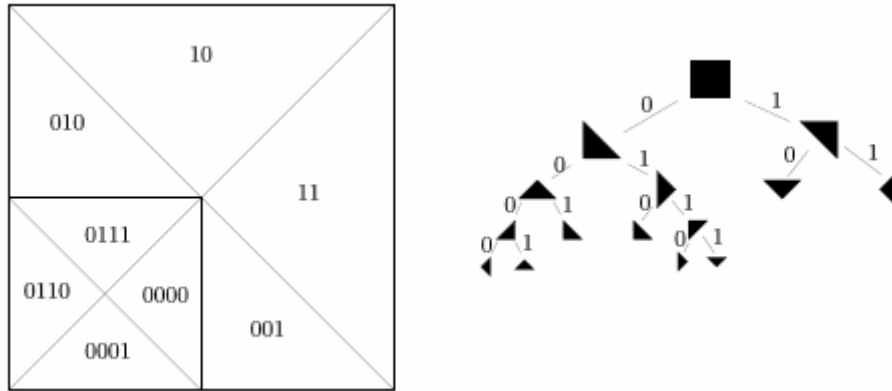


Figure 5: The binary tree representation of a RTIN partition.

Any single partition corresponds, as in figure 5, to a binary tree. Thus, a RTIN surface approximation can be represented by a binary tree and the height of each of the triangle vertices from the input elevation model. The structure of the tree allows us to calculate the (x, y) coordinates of the vertices of a triangle from its label in the tree. This is the advantage of the RTIN structure over a general TIN. Neither the coordinates nor the label need to be explicitly stored. Contrast this with the general TIN which must store the (x, y) coordinates (or indices) of the vertices of the triangles it contains. In this way, the RTIN resembles the regularly sub-sampled grid. In the sub-grid, the x, y coordinates of a point can be calculated from the point's indices in the array of height values. In the RTIN, the x, y coordinates of the vertices of a triangle can be calculated from the triangle's label. Both the point indices in the sub-grid and the triangle label in the RTIN are addresses into the structure. They are not

stored in the structure. Unlike the regular sub-grid, the RTIN allows non-uniform sampling; some regions may be more densely sub-samples than others. In this way the RTIN resembles the general TIN.

Determining the coordinates of the three vertices of a triangle from its label is straight forward. The label describes a path in the binary tree representing the surface. At each step in this path, as one descends from the root, one can construct the vertices of the left or right child triangle from the vertices of the parent. If $\Delta(v_1, v_2, v_3)$ is the parent triangle (vertices are listed in counter-clockwise order with v_3 the right-angled vertex) then the left child is $\Delta(v_3, v_1, m)$ while the right child is $\Delta(v_2, v_3, m)$ where (m_x, m_y) (the x, y coordinates of m) are the x, y coordinates of the midpoint of the line segment $v_1 v_2$ and the z coordinate (i.e. height value) of m is obtained from the input array location (m_x, m_y) . We assume that the x, y coordinates of the vertices are integers and represent indices into the input array. If not, they can be scaled and rounded to the appropriate integral range.

A straightforward implementation of this binary tree structure for a single surface approximation is inefficient in its use of space. Each node requires two pointers to its children and the height of its vertices must be stored as well. This, along with the fact that there are twice as many nodes in the tree as triangles in the approximating surface, makes the RTIN method of single surface approximation more space intensive than a TIN representation.

The advantages of the RTIN become more apparent when one wishes to represent a hierarchy of surface approximations rather than just a single approximation. In some sense, the binary tree representation of a RTIN partition is tailor-made for hierarchies since, if it represents a partition at a particular level of detail, the binary tree that corresponds to the most detailed RTIN partition of the input elevation model.

Of course, to obtain a surface approximation from the binary tree, we also need the height values of the vertices of all triangles in the approximation. For this we store the original input array of elevations.

For the price of this one RTIN surface approximation (albeit the most expensive), we obtain a representation of all RTIN surface approximations. Methods for extracting a particular surface approximation from this hierarchy are described later.

3.2 Reducing Space Usage

This section describes issues concerning the space used by the hierarchy and how that space may be reduced.

If the original input was a $2^k + 1 \times 2^k + 1$ array, the binary tree representing the hierarchy is the complete binary tree of depth $2k + 1$. Each leaf of the tree represents a triangle that is half of the input grid square. In this case, it is much more space efficient to store the binary tree as an array where the label of a node, treated as the binary representation of an integer, determines the node's location in the array. The problem of having, for example, labels 0 and 00 both representing the same integer can easily be solved by appending a 1 to the node label to form the array index. The addressing scheme obtained in this way corresponds to the typical implicit array representation of a binary heap. By using the array representation, we eliminate child pointer storage.

If the input does not have size $2^k + 1 \times 2^k + 1$ but rather size $w \times h$ (where w and h are less than $2^k + 1$ but either w or h are greater than $2^{k-1} + 1$) then a complete binary tree of depth $2k + 1$ contains some nodes that do not correspond to triangles with vertices that are grid locations. The triangles represented by these leaves do not "cover" the input array (i.e. their projection into the x, y -plane does not intersect the range of the indices of the input array). They are invalid. A more complicated indexing scheme of the implicit array representation can be used in this case. The basic idea of this scheme is to follow the implicit array representation but to calculate which of the nodes in the binary tree represent invalid regions, and to offset the implicit array index of a node by the number of invalid regions whose implicit array index precedes it. The space requirement is then reduced from being proportional to $(2^k + 1)^2$ to being proportional to wh .

Since these space saving mechanisms eliminate the need for a node to store pointers to its children, one might well ask what information is stored in the tree, since the pointer structure and the height values were the only pieces of information required to represent a single surface approximation.

The only vital piece of information that must be stored in the nodes of the hierarchy is an indication of whether or not the triangle is part of the approximation surface. In the single surface approximation, the fact that a node was a leaf of the binary tree indicated that it represented a region in the approximation. In the hierarchy, which is used to represent many different approximations, this can be accomplished using a single bit per node which indicates whether or not the node's triangle is a part of the approximation.

The total storage of a $(2^k + 1) \times (2^k + 1)$ image is the amount used for the array of height values $((2^k + 1) \times (2^k + 1)$ words) plus one bit for every node in the tree ($2^{2k+1} - 1$ bits).

This representation, however, is too sparse to be of much use for any algorithm which operates on the surface approximation. One might reasonably demand that each node contain some representation of the error of its triangle in order to determine how well the approximation fits the true surface, and in order to calculate appropriate single approximations from the hierarchy.

One may also demand the ability to “walk” from one triangle of the surface to a neighboring triangle of the surface. That is, given the address of a triangle, calculate the addresses of the (at most) three triangles that share an edge with it. We describe in the next section how this may be accomplished using an additional three bits of storage per triangle.

3.3 Neighbor Calculation

Often the algorithms one needs to execute on the approximation surface require the ability to traverse the surface from triangle to adjacent triangle. For example, one algorithm for calculating the watershed of a stream performs a hill-climbing operation which follows a path of steepest ascent from triangle to adjacent triangle. In order to accomplish this, the algorithm needs to determine the neighbors of any given triangle. In particular, one needs a function that, given the label of a triangle, can calculate the label of its adjacent triangles.

In order to describe this function let us number the vertices of a triangle in counter-clockwise order from 1 to 3 so that vertex 3 is the right-angled vertex. Define the i -neighbor of a triangle as the neighbor that does not share the triangle's vertex i (see figure 6).



Figure 6: The i -neighbors of a triangle. In this case, all neighbors are the same size as the triangle. The numbers within the inner triangle are the vertex numbers of the inner triangle.

The same-size i -neighbor of a triangle t is the triangle's i -neighbor in the uniform partition that contains t . The same-size neighbor may or may not be a part of the approximation surface. The function which finds the i -neighbor of a triangle first finds the same-size i -neighbor of the triangle and then uses some extra information stored with the triangle to determine the true i -neighbor. The extra information is a bit which says whether or not the i -neighbor is the same size as the triangle or not. If the bit indicates that the i -neighbor is not the same size as the triangle then the i -neighbor must be smaller if $i = 1, 2$ or larger if $i = 3$. Alternatively, one may avoid the extra 3 bits of storage by examining the surface bit of the same-size neighbor and its parent (or child).

The following recursive equations calculate the labels of the same-size neighbors of a triangle specified by its label. $N_i(t)$ is the same-size i -neighbor of t . The symbol λ is the (empty) label of the root. The symbol \emptyset represents "no neighbor". The function \circ is concatenation with the understanding $\emptyset \circ 0 = \emptyset \circ 1 = \emptyset$

$$\begin{aligned}
N_1(\lambda) &= N_1(0) = N_1(1) = \emptyset & N_2(\lambda) &= N_2(0) = N_2(1) = \emptyset & N_3(\lambda) &= \emptyset \\
N_1(p0) &= N_3(p) \circ 1 & N_2(p0) &= p \circ 1 & N_3(0) &= 1 \\
N_1(p1) &= p \circ 0 & N_2(p1) &= N_3(p) \circ 0 & N_3(1) &= 0 \\
& & & & N_3(p0) &= N_2(p) \circ 1 \\
& & & & N_3(p1) &= N_1(p) \circ 0
\end{aligned}$$

An example of the use of these equations is given in Figure 7.

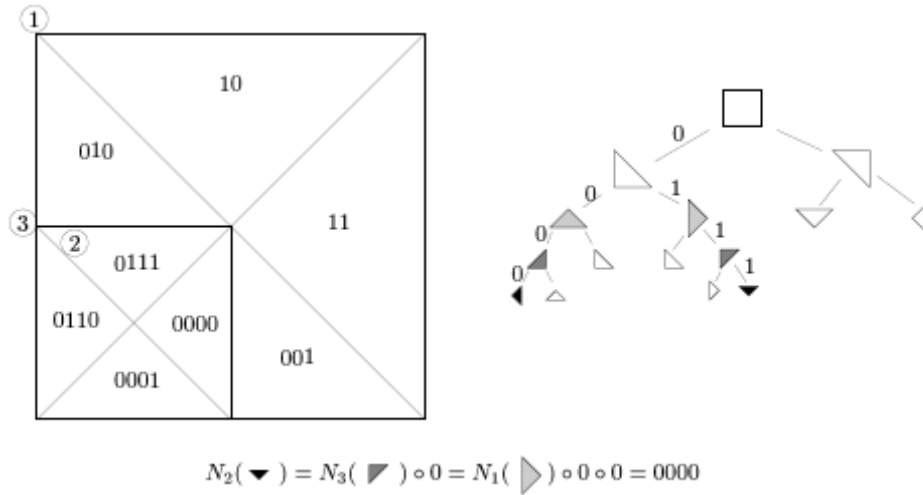


Figure 7: Use of the recursive neighbor equations to calculate the 2-neighbor of 0111.

A recursive algorithm based on these equations is fast enough in nearly all situations. Nonetheless, it may be of interest to note that these calculations may be performed using a small number (unrelated to the length of the label) of arithmetic and bitwise logical operations, provided the label is short enough to fit in a computer word (see the C-code in Figure 8).

The code works by first reducing the problem to calculation $N_3(t)$; the 1-neighbor, or 2-neighbor of a triangle t is simply the parent of the 3-neighbor of a child of t . From the recursive equations, if $t = uvw$ where v is either 00 or 11 and w is in the regular language $[01,00]^*$ then $N_3(t) = \overline{uvw}$ where \overline{x} is the bitwise compliment of x . This is the property exploited by the sample C-code. Complimentation of vw is accomplished by a single addition.

3.4 On-the-fly Surface Approximation

Using the RTIN hierarchy defined in the previous section, we can create an algorithm that quickly constructs a surface approximation. Since the application we have in mind is interactive visualization, the speed at which the approximation can be constructed is of prime importance. The construction time together with the rendering time determines the number of scenes that can be drawn per second.

There are, essentially, two approaches to constructing an approximation from a hierarchy: bottom-up and top-down. Bottom-up starts with the finest, or most accurate, approximation and progressively relaxes it – letting a large triangle, for example, replace smaller triangles if the large triangle well-approximates the smaller ones. Top-down starts with the coarsest approximation and progressively refines it. Bottom-up works well in cases where the approximation is close to the finest approximation, but it must perform many relaxations in order to reach a coarse approximation. Top-down quickly reaches coarse approximations, but may waste time refining very coarse approximations if the desired approximation is very fine.

```

int neighbor[3][4] = {{0, 0, 0, 0}, {0, 0, 0, 0}, {0, 0, 2, 1}};
#define SIEVE0 0xAAAAAAAAAAAAAAAA
#define SIEVE1 0x5555555555555555

int sameSizeNbr(int t, int i)
/* return the label of the same size i-neighbor of t. */
{
    unsigned int a,b,c,n;

    if (t <= 3) return neighbor[i-1][t];
    if (i == 1) t = (t << 1) | 1; /* reduce to 3-neighbor calculation */
    else if (i == 2) t = (t << 1); /* reduce to 3-neighbor calculation */
    c = ((t << 1) ^ t) & SIEVE0;
    a = c | (c << 1) | 1;
    b = (a + 1) ^ a; /* most of the work is done by this addition */
    if (b < (t>>1)) {
        n = t ^ b;
    } else if (((t & SIEVE0) << 1) > t) {
        if (((t & SIEVE1) << 2) > t) {
            n = t ^ (b>>1);
        } else {
            n = t ^ (b>>3);
        }
    } else {
        return 0; /* 0 means "no neighbor" */
    }
    if (i != 3) return (n >> 1);
    else return n;
}

```

Figure 8: C-code for fast triangle neighbor calculation

We use a top-down approach because the amount of work done to construct an approximation in a top-down fashion is proportional to the number of triangles in the approximation. Thus, reducing the number of triangles in the approximation speeds up the calculation of the approximation, as well as decreasing the time to draw the approximation. This assumes that the time to draw an approximation decreases as the number of triangles in the approximation decreases. This assumption is supported by our experimental results.

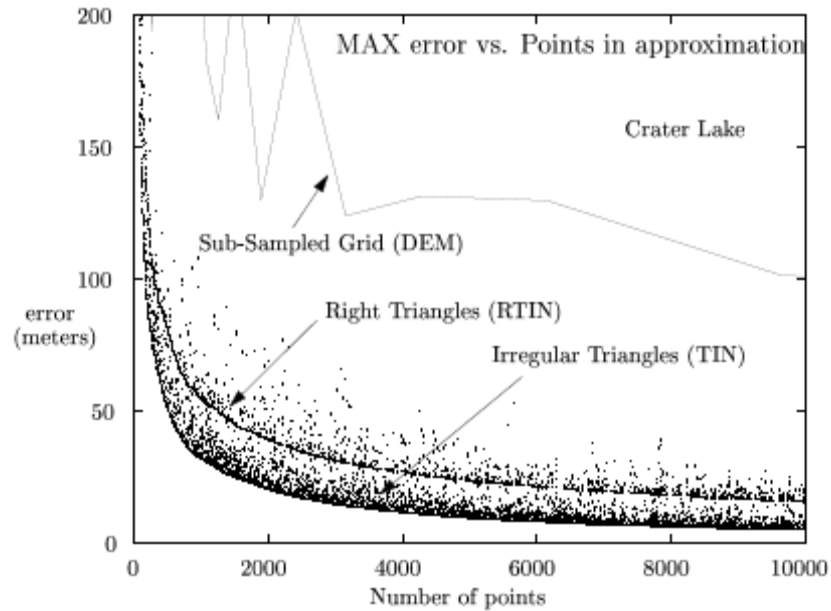
Given our choice of top-down method, we must decide how to choose the triangles to refine. Again, there are essentially two choices: thresholding or prioritized refinement (though many variations of these two themes are possible). Prioritized refinement chooses to refine the approximation in the most poorly approximated parts of the surface until the approximation contains a certain number of triangles. The single approximation results presented use a version prioritized refinement that is explained there in detail. On the other hand, thresholding insists that the approximation everywhere satisfies some constraint.

The constraint may be as simple as that every point in the input data lies within ε of the approximation, or it may be a more complicated constraint based on the eye position a desired visual fidelity. In any case, the number of triangles produced in the approximation is not as tightly controlled as in prioritized refinement. A surface may require few triangles to achieve the desired overall fidelity in certain cases, while requiring many in other cases. Certainly, by decreasing the overall requirement, the number of triangles needed in the approximation decreases, but the control is not as precise as in the previous case.

Either method may be used to achieve surface approximations. Here we outline a method based on thresholding that is used in our implementation of an interactive visualization system. An initial preprocessing phase allocates to each triangle a measure of how accurately it approximates the surface – the triangle's error. In our implementation this is the maximum over points "covered" by the triangle of the vertical distance from the point to the triangle. The preprocessing phase also initializes the RTIN hierarchy. After its completion, a triangle's error need never be recalculated. Other triangle specific information may be calculated and stored in the hierarchy at this time, for instance, the triangle's surface normal, its display properties, etc. This increases the space requirements of the hierarchy but may decrease the display time.

The heart of the algorithm is an augmented depth-first search of the hierarchy that updates the surface approximation in response to movements of the eye position. The depth-first search starts with the approximation being the two triangles represented by the two children of the root of the hierarchy (the binary tree representing the finest RTIN partition). It visits each child of the root. In general, if the search is visiting a node and the triangle that the node represents is not sufficiently close to the surface, then the depth-first search splits the triangle (which may cause many triangles to split recursively in order to avoid gaps in the surface) and then recursively visits the children of the node. The recursion caused by a triangle split may split triangles whose nodes have not yet been visited by the depth-first search. If the depth-first search visits a node whose triangle has been previously split, it recursively visits the node's children.

The decision of whether a triangle is "sufficiently close to the surface" is based on both the error of the triangle and the distance of the triangle from the eye position. In general, one may choose an inverse-distance weighting of the triangle's error, the area that the projection of the triangle occupies on the display screen, or other measures. One obvious common criteria is that triangles at the finest level of detail should not be refined.



**Figure 9: Maximum error versus number of points in approximation.
Crater Lake data set.**

The precise function used to determine if a triangle is “bad” is not our main focus. We present results that call a triangle bad if the inverse weighting of its error is greater than some threshold; or if it is large and close to the eye position, e.g., within 700 meters of the eye position, any triangle that is not at the finest level of detail is bad. A more appropriate measure for interactive visualization might also include some measure of the visibility of the triangle. For instance, triangles outside the field of view, no matter how poorly they approximate the surface, are not seen by the observer and, hence, should not be bad.

Once the depth-first search creates the approximation, a second depth-first search traverses the hierarchy and draws the triangles that form the approximation.

3.5 Experimental Results

3.5.1 Single surface performance

To construct a single surface approximation, we use a procedure similar to the general TIN construction method of Fowler and Little from 1979. The procedure is iterative and greedy. In each iteration, the point least well represented by the current approximation is added to the set of vertices, and the x, y projection of the set is retriangulated (using a Delaunay triangulation) to form the next approximation. In our case, we do not have the flexibility of adding an arbitrary

point to the current approximation. We can refine an approximation only by splitting a triangle that is in the approximation. However, we can, in spirit, mimic this greedy approach, by splitting the triangle that contains the least well approximated point, perhaps causing other triangles to split. To implement this, we maintain a priority queue of the triangles in the approximation prioritized by their error and incrementally split the worst triangle.

For both TIN and RTIN construction, the procedure attempts to reduce the maximum error in the surface approximation at each iteration. It targets the point which has the current worst error, or the triangle which contains this point in the case of RTINs. If, however, instead of desiring a small maximum error, one desires a small RMS (root mean squared) error, then the analogous procedure is to find the triangle with the worst RMS error or the point whose deletion most decreases the RMS error. For TINs, finding the point that most decreases the RMS error is prohibitively expensive. However, for RTINs, this greedy heuristic is rather simple to adopt: at each iteration the triangle with the maximum RMS error is chosen to be split.

As one would expect, the maximum error in the surface decreases more rapidly as a function of the number of points in the approximation when we allow general triangles in the partition (the TIN approximation) as opposed to the more constrained RTIN approximation. A sub-sampled DEM is the worst performer under this measure (figure 9). Note that we used linear interpolation to calculate the surface in each case. The plots of the RTIN and TIN errors are scatter-plots. The points cluster so densely that they present the illusion of smooth curves. The plots show that, occasionally, adding a point to an approximation increases the maximum error. This is a result of using a non-optimal algorithm to calculate the RTIN and TIN approximations.

We obtain a similar result when we consider the RMS error as a function of the number of points in the approximation. In this case, the RTIN algorithm chooses to split the triangle which has the largest RMS error (figure 10).

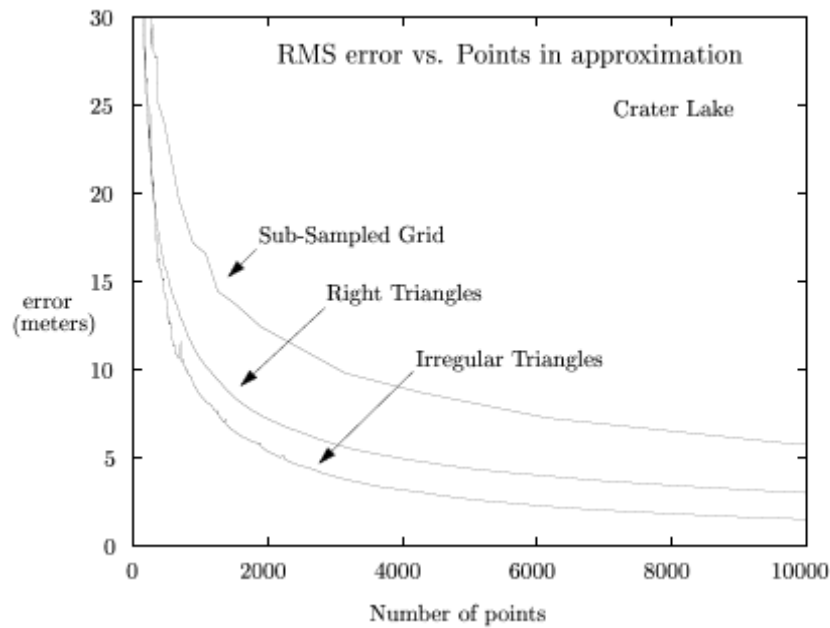


Figure 10: Root mean squared error versus number of points. Crater Lake data set.

Even when we consider the maximum error as a function of the amount of memory required to store the approximations, the irregular triangulation or TIN still outperforms both the RTIN approximation and the gridded DEM approximation (figure 11). We count only the space for the approximation itself, and we assume that the coordinate values can be stored in two bytes.

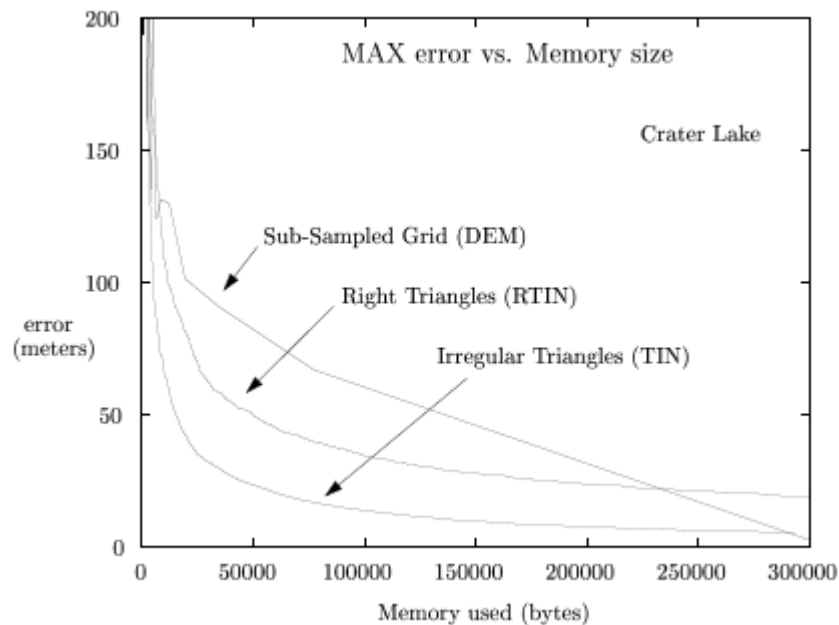


Figure 11: Maximum error versus memory usage.

In the case of the TIN, the amount of memory used is approximately 30 bytes per vertex in the approximation. This counts the two bytes each for the x , y and z coordinates and four bytes each for the, on average, six neighbor pointers of each vertex.

For the RTIN, the storage required for each node in the binary tree is two bytes to hold the height of the midpoint of the hypotenuse and eight bytes for the two child pointers. This is a single surface approximation; the leaves of the tree represent triangles in the approximation. Thus, we do not need to store neighbor information in the nodes. To calculate the neighbor of a triangle, we calculate its same-size neighbor and take the closest leaf to that neighbor (the neighbor itself, its parent, or its child). The number of triangles in the approximation (i.e. leaves in the binary tree) is approximately twice the number of vertices in the approximation. The total number of nodes in the tree is approximately twice the number of leaves. Thus the RTIN uses approximately 40 bytes per vertex in the approximation.

For the gridded DEM, the storage requirement is approximately two bytes per point in the approximation, since only the height values need to be stored.

The one instance in which the TIN approximation does not achieve the best performance of the three is when the RMS error of the approximation is considered as a function of the memory used. In this case, the gridded DEM achieves the best error for a given memory size (figure 12). This result supports Kumler's observations and, indeed, his error measures are similar to the RMS error measure. That the gridded DEM produces small error may be largely attributed to the fact that its triangles are on average, quite small.

The performance of the RTIN as a single surface approximation does not match the performance of the general TIN approximation scheme. This will always be the case when performance is measured as a function of the number of points in the approximation. There is some hope, however, that other means of storage can bring down the memory requirements of the RTIN to the point where it competes with the TIN on single surface approximations when performance is a function of the memory used. For example, the labels of the triangles in the approximation may be hashed rather than stored in a binary tree. This avoids allocating space for internal nodes that are extraneous in a single surface approximation.

The data for the figures shown above are elevation data for Crater Lake West in southern Oregon. The results are similar to those obtained for other areas such as Yakima, Washington; Tucson, Arizona; and Reno, Nevada. We obtained these data sets and others from the U. S. Geological Survey. We obtained the code that was used to produce the irregular TINs from Garland and Heckbert.

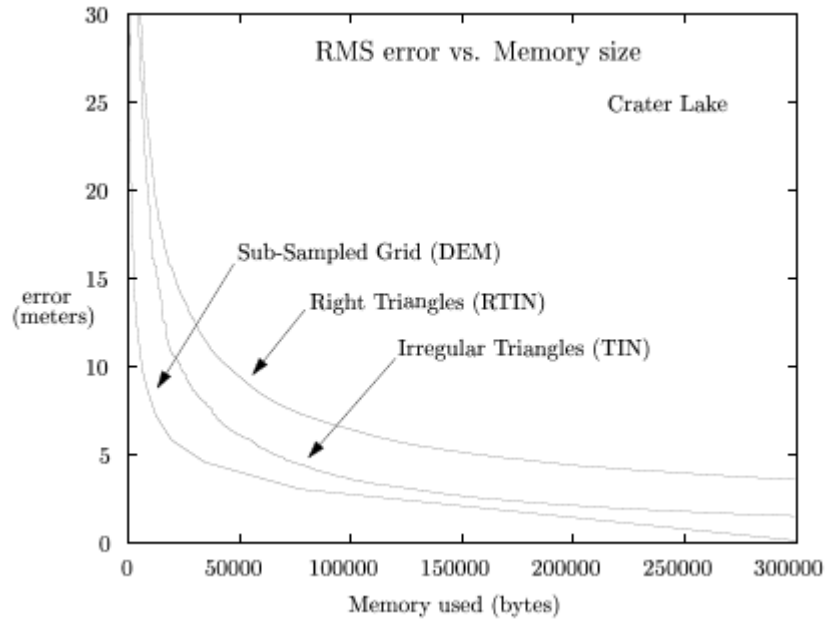


Figure 12: Root mean squared error versus memory usage. Crater Lake data set.

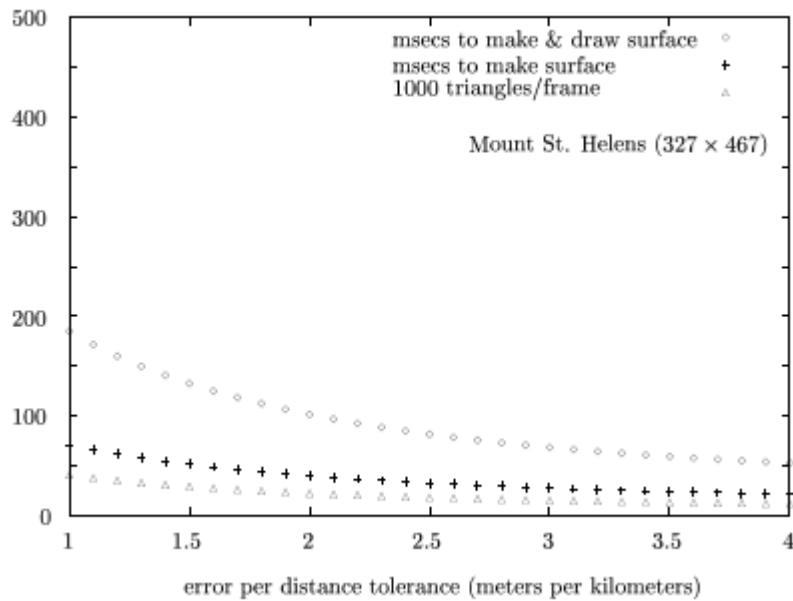


Figure 13: The average time to construct and time to construct and draw a surface approximation of Mount St. Helens as a function of the error per distance tolerance. The graph also shows the number of triangles (in 1,000s) in the approximation.

3.5.2 Hierarchical Performance

The rather poor performance of the RTIN structure in describing a single approximation is in contrast to its performance as a hierarchy for achieving multiple levels of detail in an interactive environment. An interactive visualization system, called TopoVista, incorporating the ideas presented in this report may be downloaded from the TopoVista web site. All experimental results presented in this section are from this implementation.

The system on which we ran these experiments is an SGI Indigo2 with a High Impact graphics board and 128 megabytes of main memory, running IRIX 6.5. The program uses the OpenGL graphics package and OpenGL Utility Toolkit.

We present results for a small elevation model (figure 13), Mount St. Helens (327 x 467 data points), and a larger elevation model (figure 14), Mount Rainier East (977 x 1405 data points). In both cases, we measured the time to construct a surface approximation, the time to both construct and display the approximation, and the number of triangles in the approximation. We obtained these numbers by calculating the total time (and number of triangles drawn) to move the eye position around the perimeter of the elevation model, looking toward the center, and then dividing by the number of approximation calculated during this circuit. We repeated the experiment for varying error per distance tolerances. The tolerance specifies the vertical discrepancy, in meters, between the

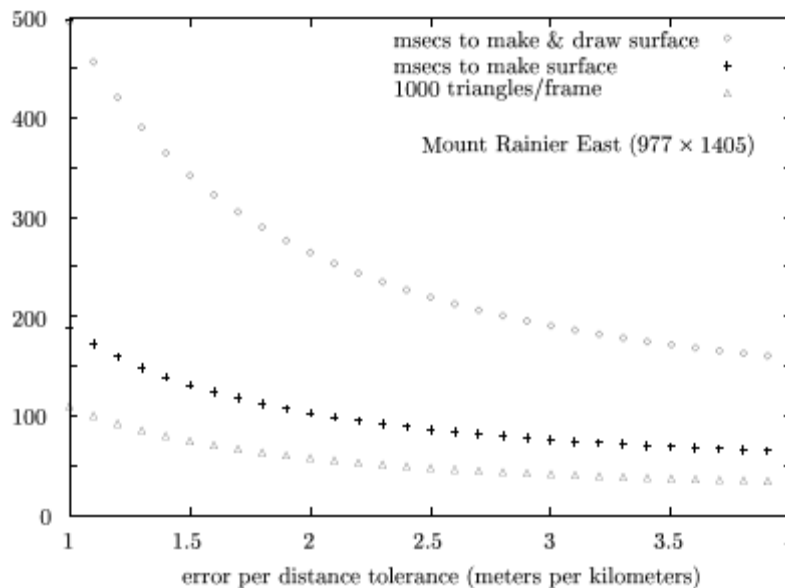


Figure 14: The average time to construct and time to construct and draw a surface approximation of Mount Rainier East as a function of the error per distance tolerance. The graph also shows the number of triangles.

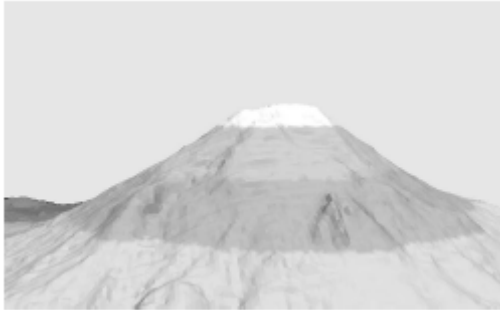


Figure 15: Full detail, pre-1980 Mount St. Helens (305K triangles).

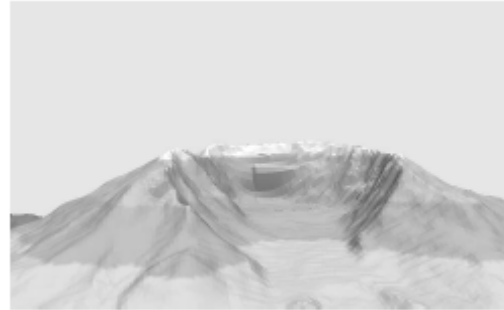


Figure 16: Full detail, post-1980 Mount St. Helens (305K triangles).

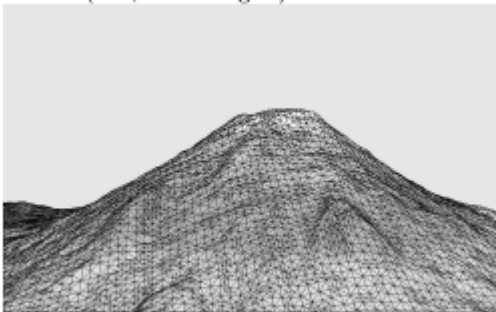


Figure 17: Tolerance=1m per km. (44,500 triangles).

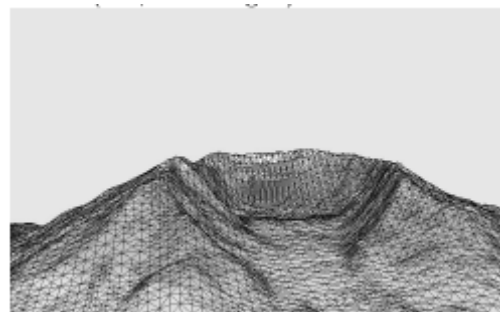


Figure 18: Tolerance=1m per km (39,800 triangles).

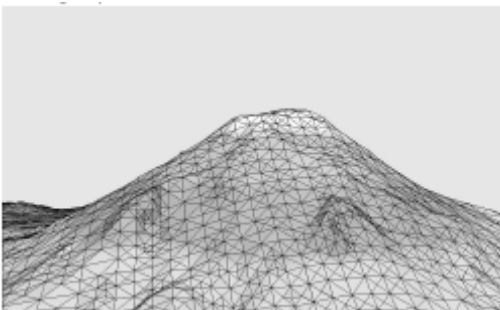


Figure 19: Tolerance=4m per km. (10,600 triangles)

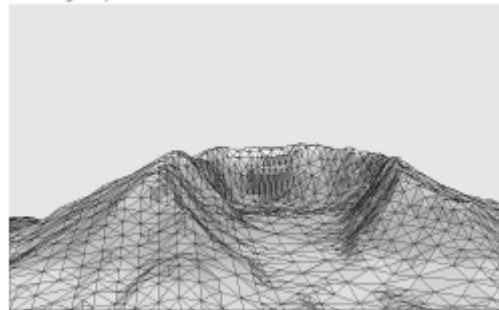


Figure 20: Tolerance=4m per km. (10,200 triangles)

approximation and the original elevation model that is acceptable per one kilometer distance from the observer. Thus, an error per distance tolerance of 2.1 means that a triangle that is two kilometers from the observer, and within their field of view, must have a vertical discrepancy of at most 4.4 meters. In addition, in both cases, we insist that the approximation be at full detail within the observers' field of view to a distance of 700 meters.

Allowing an error per distance tolerance of 1.0 within the observer's field of view resulted in approximately 7.5 (Mount St. Helens) or 25 (Mount Rainier) times fewer triangles than are in the triangulation of the original data points. With the

tolerance at 4.0, the decrease was by a factor of 26 (Mount St. Helens before May 18, 1980 eruption (figures 17 and 19) and after (figures 18 and 20), as well as the full detail versions of the same views.

3.6 *Comparison to Previous Work*

Structures, similar to the RTIN structure, that support multiple approximations with multiple levels of detail have been discovered independently by Lindstrom, Koller, Ribarsky, Hodges, Faust, Turner and Popeo.

Lindstrom describes an interactive visualization system whose fundamental data structure is a hybrid between a RTIN and a sub-grid. The surface is partitioned into rectangular blocks each of which is regularly sub-sampled (like a sub-grid) then, within each block, a bottom-up reduction process eliminates vertices (data points) that are sufficiently well-approximated, forming a RTIN-like surface of right-angled isosceles triangles. Special care is taken at block boundaries to insure that no gaps are created. Many details of the data structure are left open, but space usage of between six and 28 bytes per data point are quoted, including two bytes for the input elevation. This compares favorably with our pure RTIN approach that requires between ten and 14 bytes per data point. The difference in space usage between the two methods can be attributed to the fact that the RTIN hierarchy contains approximately four times as many triangles as data points. We store an error measure (and neighbor information) for each triangle while Lindstrom stores a single error measure for each data point. As a result, we can guarantee that our surface approximation is within a given tolerance (in our case, vertical discrepancy) of the original elevation model, while Lindstrom cannot. It should be noted that both our ten byte version and Lindstrom 's six byte version compress the surface error from its full detail two-byte representation to a one-byte representation, incurring a loss of precision.

Popeo describes an RTIN-like tree structure as a special case of a more general framework of multi-resolution models. He gives no experimental performance results for these structures, but does list, without detail, some of the operations, such as neighbor calculation, that such a structure should support.

3.7 *Conclusions*

In this report, we present a type of surface approximation that is a restricted form of a triangulated irregular network; it consists solely of right-angles isosceles triangles, and is therefore called a right-triangulated irregular network or RTIN. We describe how a RTIN may be stored efficiently, and yet still support operations such as traversal from triangle to adjacent triangle that are required by most applications operating on surfaces.

The performance of RTINs for single surface approximations is close to that of TINs but, as one might expect, the accuracy achieved by a RTIN as a function of the number of points in the approximation is less than the accuracy of a TIN. Rather more surprising and disappointing is that the performance of a RTIN is worse than a TIN as a function of the memory used by the approximation. More efficient storage schemes for RTINs are possible and further research is needed to see if such RTIN representations can out-perform TINs.

RTINs perform much better as a representation of many approximations in a hierarchy that supports multiple level-of-detail approximations. The storage scheme we describe for a single surface approximation actually contains all coarser RTIN approximations. We describe a method for extracting an appropriate approximation quickly and illustrate its use in a system, TopoVista, that permits interactive visualizations of USGS DEM data.

4.0 Mirage NFS Router

Mirage is a system that aggregates multiple NFS servers into a single, virtual NFS file server. It is interposed between the NFS clients and servers, making the clients believe that they are communications with a single, large server. Mirage is an NFS router because it routes an NFS request from a client to the proper NFS server, and routes the reply back to the proper client. Mirage also prevents Denial of Service (DoS) attacks on the NFS protocol, ensuring that all clients receive a fair share of the servers' resources. Mirage is designed to run on an IP router, providing virtualized NFS file service without affecting other network traffic. Experiments with a Mirage prototype show that Mirage effectively virtualizes an NFS server using unmodified clients receive a fair share of the NFS server even during a DoS attack. Mirage imposes an overhead of only 7% on a realistic NFS workload.

4.1 *Mirage Introduction*

Mirage provides NFS clients with the illusion that a collection of NFS servers is actually a single, virtual NFS server. Mirage is not an NFS server. It is an NFS router, which operates by routing client requests to the appropriate NFS servers, and routing replies back to the appropriate clients (Figure 21). Mirage exports a set of file systems that is the union of the file systems exported by the NFS servers. Clients mount a file system from the virtual server and access its contents as if Mirage were a real, physical NFS server. Mirage is thus fully transparent to the clients and servers, allowing unmodified client and server implementations to use Mirage.

An NFS file system is stored on a single NFS server. NFS has no provision for spreading a file system over multiple servers. Due to this limitation, as the

system scales to more users, the NFS server will become overloaded with the increased demand. The system administrator has few solutions to this problem, including upgrading the NFS server or introducing an additional server and moving some files from the old server to the new. The former solution is expensive and disruptive, while the second requires reconfiguring the clients to be aware of the new server. Similarly, if an NFS server runs out of storage, the only way to gain additional space is to upgrade or add an additional server.

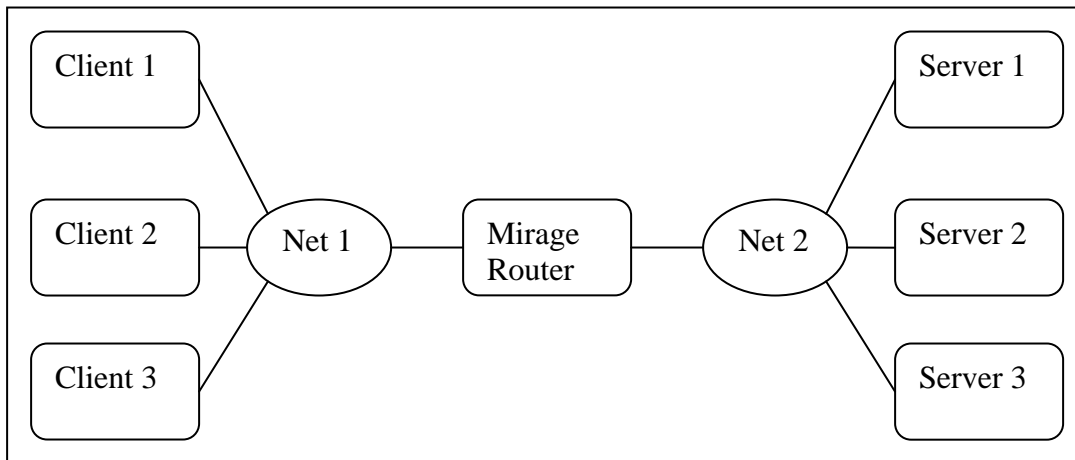


Figure 21: Mirage routes requests and replies between the NFS clients and the NFS servers.

Mirage implements a virtual server abstraction that solves these system administration problems. Clients perceive a single NFS server, unaware that it is actually a virtual server that is the aggregation of the real NFS servers. Files and file systems can be moved from one NFS server to another without reconfiguring the clients. Similarly, new NFS servers can be deployed without the clients' knowledge. All that is required is to reconfigure Mirage to include the new server in its visualization. By hiding the server configuration details from the clients, Mirage allows NFS to scale to large numbers of servers without an overwhelming increase in system administration complexity.

Mirage virtualizes an NFS server such that the IP address of the virtual server is the IP address of the Mirage router. Client NFS requests are delivered to the Mirage router, which rewrites the requests and forwards them to the appropriate NFS server. The clients are unaware that Mirage is a router, or that it is aggregating multiple NFS servers into a single virtual server. No client modifications are necessary, which is advantageous because the NFS client protocol is usually implemented as an integral part of the host operating system on the client computer. To modify the NFS implementation would require installing a custom kernel on the client computer, a task that is beyond the abilities of most end users, and inconvenient for most system administrators.

Mirage also supports unmodified servers. Server modification is infeasible because many NFS servers are commercial products and contain proprietary code. Commercial servers also provide additional features such as snapshots, which not only are complicated to implement, but protected by patents. Furthermore, since Mirage doesn't require server modifications and exists outside of the server, it is always possible to access the NFS servers directly if suffers a failure or is otherwise unavailable.

Mirage contains support for preventing DoS attacks on the NFS protocol. A DoS attack is one in which a malicious client overwhelms the server with requests, preventing legitimate clients from accessing the server. These attacks may not be detected by existing DoS mechanisms because they may generate very little network traffic, yet induce a high load on the server. The request might be quite small, but require the NFS server a lot of effort to process. Mirage mitigates DoS attacks by ensuring that each NFS client receives a fair share of the servers' resources.

Mirage is designed to run on a programmable network router. It has a minimal amount of state, and is able to recover its state automatically after a crash. Mirage operates by re-writing packet contents based on table-lookups, and therefore introduces a minimal amount of overhead to the packet processing. We have developed two prototype implementations, one as a user-level NFS proxy, and another as a Linux kernel module. The kernel module version has only a 7% slowdown of a large compilation benchmark over that of a simple IP router. We believe this to be an acceptable overhead for the benefits that Mirage provides.

4.2 NFS

Mirage implements the NFS protocol, version 2. The NFS protocol is based upon SUN Remote Procedure Call (RPC), and therefore uses a request/reply paradigm for communication between the clients and the servers. The protocol uses handles to represent files, directories, and other file system objects. An NFS handle is a 32-byte quantity that is generated by the server and opaque to the client. The client receives the handle for the root directory of a file system when it mounts that file system. The mount request includes the name of the file system to be mounted (the mount point). The NFS server checks the mount request for the appropriate security and authentication and then issues a reply containing the handle for the root directory of the file system. A client uses a handle to access the contents of its associated object, as well as to obtain handles for additional objects.

An NFS client obtains a handle for a desired object in an iterative fashion. It works its way through the desired pathname one component at a time, sending a lookup request to the server containing the handle of the current directory and

the name of the desired component to the server, and receiving back a handle for the component. For example, to get a handle for the file /foo/bar, the client first sends a lookup request containing the handle for /foo and the string “bar”, and receives back the handle for /foo/bar. The client then uses the handle to read and write the file. Figure 22 illustrates the sequence of events required to read the file /home/fred/photo.

Client Request	Server Reply
Mount("/home")	handle _{home}
Lookup(handle _{home} , "fred")	handle _{fred}
Lookup(handle _{fred} , "photo")	handle _{photo}
Read(handle _{photo} , 0-1024)	First 1024 bytes
Read(handle _{photo} , 1024-2048)	Next 1024 bytes

Figure 22: Sequence of NFS client requests and server replies to read the first 2048 bytes of /home/fred/photo.

The use of handles in the NFS protocol poses two problems for Mirage. First, the NFS server generates handles however it desires, as long as different objects have different handles. Typically, the NFS server will encode information in the handle that identifies the location of the object in the server’s internal file system, improving access performance. The handles are opaque to the client, however, so it is unaware of what information the handle holds. Since Mirage virtualizes multiple NFS servers into a single server, it must ensure that different objects have different handles, even if the objects reside on different NFS servers. Mirage has no control over how the servers create their handles, so there is no way for Mirage to ensure that the handles generated by the servers do not conflict. As a result, Mirage must virtualize the NFS handle space by creating its own virtual handles and the physical handles used by the servers. This requires Mirage to maintain state about the mapping, increasing Mirage’s complexity.

The second handle-related issue that affects Mirage is once a client has a handle, it may use the handle indefinitely to access the associated object. The client never needs to refresh the handle, therefore the server cannot subsequently change the handle contents. If the server crashes and reboots, it may no longer have information about handles issued before the crash, causing it to return “stale handle” errors to the client. This causes the client to re-lookup the handle, if possible. Handle longevity is an issue for Mirage because it means that not only must Mirage maintain the mapping from a virtual handle to a physical handle indefinitely, but Mirage must also be able to reconstruct that mapping after a crash. If Mirage did not reconstruct the mapping, then Mirage would have to return a “stale handle” error to the client, causing Mirage to no longer be transparent to the client.

4.3 *Mirage*

Mirage addresses the NFS scalability problem by virtualizing multiple NFS servers as a single NFS server. This virtualization is entirely transparent to the client and the servers, so the clients are not aware they are dealing with a collection of server. There are several issues that Mirage must resolve, including the abstraction presented to the clients, how virtual handles are maintained and validated, and how Mirage recovers from crashes.

4.4 *Virtual Server Abstraction*

There are at least two possibilities for the virtual server abstraction presented by Mirage. The simplest is that the set of mount points exported by the virtual server is the union of the sets of mount points exported by the underlying NFS servers (Figure 23).

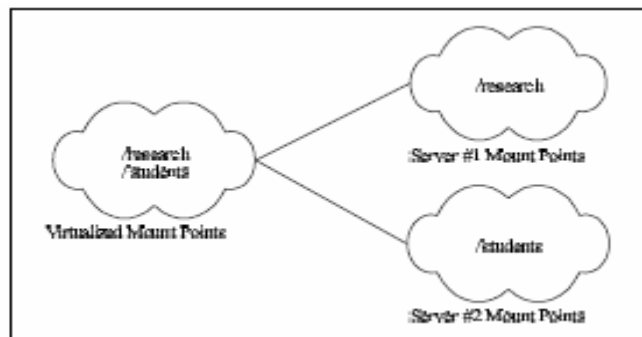


Figure 23: The set of mount points exported by the Mirage virtual server is the union of the sets of mount points exported by the individual NFS servers.

When the Mirage router receives a mount request it forwards the request to all of the NFS servers that it virtualizes. The server that exports the desired mount point will return its root handle, while the other servers return errors. Mirage generates a virtual handle, associates it with the root handle provided by the server, and returns it to the client. In this way the client can access the entire union of the mount points and believes it is communication with one larger server. The advantage of this approach is that it requires a relatively small amount of state and processing on Mirage to implement. The Mirage router must simply associate different virtual handles with different servers, so that requests are directed properly. When a new virtual handle is created via a lookup on an existing handle, the new handle is associated with the same server as the existing handle.

The downside of this approach is that the sets of mount points exported by the servers must be disjoint. Suppose two servers export mount points with the same name. What should the virtual server export? An alternative virtual server abstraction that addresses this issue is to export the union of the file system namespaces (Figure 24). For mount points that overlap, the virtual server exports a single mount point whose name space merges the name spaces of the individual mount points. When the client mounts an overlapping mount point, subsequent accesses to that file system must be directed to the proper NFS server on a case-by-case basis. This approach allows for more flexibility in the organization of the underlying servers, but increases the virtual server's complexity because it must map virtual handles to NFS servers individually. If a new virtual handle is created using a lookup on an existing handle. It is possible that the new handle should be associated with a different server than the existing handle. Furthermore, this approach introduces semantic problems if there are name conflicts in the underlying file systems. Suppose a directory on one NFS server has the same name as a file on another server. What type of object should the clients see? It isn't obvious that there is a correct answer to this problem.

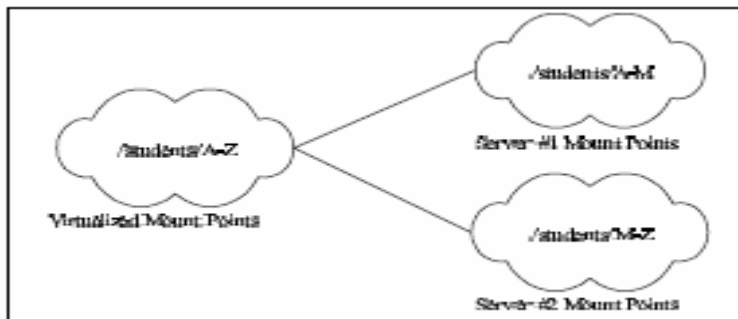


Figure 24: The virtual server exports file systems whose namespaces merge the namespaces of the underlying exported file systems.

The current Mirage prototype uses the former approach of exporting the union of the mount points exported by the NFS servers. This approach is simpler to implement, demonstrates most of the advantages of a virtual NFS server, and doesn't have the semantic problem of naming conflicts within file namespaces. We are currently experimenting with the "union of file namespaces" approach and have a partial implementation in Mirage. Its viability and value is an area of future work, however.

4.5 Virtual Handle Mapping

One of the core functions of the Mirage router is to map between the file handles produced by the Mirage router and the file handles produced by the NFS servers (Figure 25). The clients cannot be presented with the NFS server handles directly

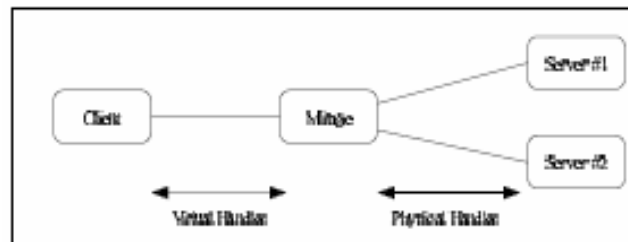


Figure 25: The Mirage router maps between the virtual file handles used by the clients and the physical file handles used by the NFS servers.

Because there is no guarantee that the NFS servers won't generate the same handle for different objects. For this reason, Mirage generates its own file handles that uniquely identify objects. We refer to the file handles generated by Mirage as Virtual File Handles (VFH), and those generated by the NFS servers as Physical File Handles (PFH). Mirage stores the mapping between a VFH and a PFH in a memory-resident handle table. When Mirage receives an NFS request from a client, Mirage looks up the VFH in the handle table to determine the proper server and PFH for the request. Mirage then rewrites NFS requests using the PFH and forwards it to the server.

Mirage must perform a reverse mapping on NFS replies. Each PFH in a reply must be mapped to the appropriate VFH before the reply is forwarded to the client. The most common reply to contain a PFH is the reply to the Lookup request that is used to resolve a file name into a file handle. Mirage looks up the PFH contained in the reply in the handle table and rewrites the NFS reply with the correct VFH before forwarding the reply to the client. If this is the first time the PFH has been used, then Mirage generates the new VFH and stores it in the handle table.



Figure 26: Format of a virtual file handle. Offsets are in bytes.

The handle table is a concern because it represents state on the Mirage router that consumes memory resources and must be recovered after a crash. It also requires processing to look up handles in the table. Mirage minimizes the state

and processing resources of the table by encoding information in the VFH (Figure 26). The VFH is a 32-byte quantity that is opaque to the client, make all 32 bytes available for Mirage's use. Mirage encodes the following information in the VFH:

Virtual Inode Number (VIN): Every object has a unique VIN.

Server ID (SID): The SID identifies the NFS server that stores the object associated with the VFH.

Handle Version (HVER): The handle version number allows multiple handles to be issued for the same object. This is used by the DoS prevention mechanism to invalidate compromised handles.

Physical Inode Number (PIN) and Physical File System ID (PFS): The PIN is the inode number of the object as assigned by the NFS server, and the PFS is the file system ID on the server. These two fields thereby uniquely identify an object within the physical file server and are used during recovery. The triple (SID, PIN, PFS) uniquely identifies an object within all of Mirage.

Mount Checksum (MCH): The MCH is a 32-bit checksum of the name of the mount point associated with the VFH. The MCH speeds up Mirage recovery.

Handle Validation Checksum (HVC) is a cryptographically secure checksum that includes the other fields of the VFH and prevents clients from forging VFHs.

4.6 *Mirage Crash Recovery*

If a traditional IP router fails, the Internet routes packets around it. This process is transparent and the communicating parties are not aware that their traffic has shifted from one router to another.

4.7 *Conclusion*

Mirage is an NFS router that provides a virtual NFS server abstraction, aggregating the resources of multiple unmodified NFS servers for use by unmodified clients. Both the clients and the servers are unaware of Mirage's presence, yet Mirage allows files and file systems to be moved between NFS servers, and new NFS servers to be deployed, without reconfiguring the clients. In this way it greatly simplifies NFS system administration.

Mirage also protects against DoS attacks. Mirage ensures that each client is given an equal share of the servers' resources. Malicious clients can waste

server resources, but never more than their share, and thus they are unable to prevent legitimate clients from accessing the NFS servers.

Experiments with Mirage show that packet latencies are significantly higher than in a standard IP router on the same hardware, but that bandwidth is only reduced by 5-7%. A realistic workload is only 7% slower using Mirage than using the NFS server directly. Mirage's low overhead, combined with the benefits of the virtualized server abstraction, demonstrate the viability of using an NFS router to build a scalable and versatile NFS system